Content Analysis and Queries in a Sound and Music Database

Stephen Travis Pope, Pierre Roy, and Nicola Orio
CREATE Lab (Center for Research in Electronic Art Technology)
Department of Music, UCSB, Santa Barbara, California 93106 USA
{stp, pierre, nico}@create.ucsb.edu, http://www.create.ucsb.edu/Paleo

Abstract

The *Paleo* database project at CREATE aims to develop and deploy a large-scale integrated sound and music database that supports several kinds of content and analysis data and several domains of queries. The basic components of the Paleo system are: (1) a scalable general-purpose object database system, (2) a comprehensive suite of sound/music analysis (feature extraction) tools, (3) a distributed interface to the database, and (4) prototype end-user applications.

The Paleo system is based on a rich set of signal and event analysis programs for feature extraction from sound and music data. The premise is that, in order to support several kinds of queries, we need to extract a wide range of different kinds of features from the data as it is loaded into the database, and possibly to analyze still more in response to queries. The results of these analyses will be very long "feature vectors" (or multi-level indices) that describe the contents of the database. To be useful for a wide range of applications, the Paleo system must allow several different kinds of queries, i.e., it needs to manage large and changing feature vectors.

As data in the database is used, the feature vectors can be simplified. This might mean discarding spectral analysis data for speech sounds, or metrical grouping trees for unmetered music. This is what sets Paleo apart from most other media database projects—the use of complex and dynamic feature vectors and indices.

This paper introduces the Paleo system's architecture, and then focusses on three issues: the signal and event analysis routines, the use of constraints in analysis and queries, and the object storage layer and formats. Some examples of Paleo usage are also given.

1. Introduction

Most prior work in sound or music databases has addressed a single kind of data (e.g., MIDI scores or sampled sound effects), and has pre-defined the types of queries that are to be supported (e.g., queries on fixed sound properties or musical features). Earlier systems also tended to address the needs of music librarians and musicologists, rather than composers and performers. In the *Paleo* system under development at CREATE since 1996, we have built a suite of sound and music analysis tools that is integrated with an object-oriented persistency mechanism and a rapid application development environment.

The central architectural feature of Paleo is its use of dynamic feature vectors and on-demand indexing, whereby annotational information derived from data analysis can be added to items in the database at any time, and where users can develop new analysis or querying techniques and then have them applied to the database's contents on-the-fly within a query. For data that is assumed to be musical sound, this might mean performing envelope detection, spectral analysis, linear prediction, physical model parameter estimation, transient modeling, etc. For musical performance data (e.g., MIDI), this might entail extraction of expressive timing, phrase analysis, or harmonic analysis.

Paleo content is assumed to be sampled sound, musical scores, or captured musical performances. Scores and performance formats can be simple (e.g., MIDI-derived) or may contain complex annotation and embedded analysis. Paleo is specifically constructed to support multiple sets of captured musical performances (for use in comparing performance expression). This includes the derivation of basic timing and dynamics information from MIDI performances (to be able to separate the performance from the "dead-pan" score), and the analysis of timbral information from recorded sounds.

For score analysis, we use a variety of methods, including simple statistical models, rule-based analysis, and constraint derivation. Sampled sound analysis is undertaken using a suite of functions called NOLib that are written in the MatLab language and can be accessed from within the Paleo environment over the net via socket-based MatLab servers. The techniques available in NOLib include all standard time-, frequency-, wavelet modulus-domain analysis operations, as well as pitch detection, instrument classification, and sound segmentation.

The two main applications for Paleo are its use in an integrated tool-kit for composers, and in a performer's rehearsal workstation. The first set of applications will put the database at the core of a composition development environment that includes tools for thematic and sonic experimentation and sketch data management. Over the course of 1999 Paleo will be integrated into an existing composer's software framework called the "Interim DynaPiano." The second platform centers on manipulating rehearsal performance data relative to a "reference"

score (which may or may not be a "dead-pan" interpretation). Users can play into the system, and then compare their performance to another one of their own or of their teacher's. Query preparation takes place using pre-built tools such as the composer's sketch browser, or by creating direct queries in a simplified declarative query language.

The Paleo software framework is an extension of *Siren* (Pope 1997b, 1998), a Smalltalk-based programming tool kit for multimedia software that runs in the *Squeak* (Squeak 1999) implementation of Smalltalk The implementation of the Paleo database persistency and access component is based on the public-domain Minnestore object-oriented database package (Carlson 1998), which allows flexible management of data and indices. The Squeak port of Minnestore is called *SMS* (Squeak Minnestore).

Paleo applications can communicate with an SMS database server over a network, and can pass sound sample data or event streams to/from the database. We currently use a simple socket-based protocol, but plan to move to a CORBA-based distribution infrastructure in the near future.

To stress-test Paleo's analysis and query tools against a realistic-sized data set, the test contents used at the start of 1999 included over 1000 scores of keyboard music (Scarlatti, Bach, Bartok, the hymnal, etc.), several hundred "world" rhythms, the SHARC database of instrument tone analyses, 100 recorded guitar performance techniques, flute performances, and spoken poetry in five languages.

2. Paleo Architecture

In Paleo, as in Siren, music and sound data are software objects in a uniform representation called *Smoke* (Pope 1997a). Smoke can be used to represent and manipulate event-like data such as a MIDI performance of common practise notation score, or signal-like data such as a sampled sound, a DSP program, or spectral data.

In Paleo's SMS data persistency layer, Smoke objects are stored in *object sets*, which are akin to database tables. Each object set stores one kind of objects, and can have any number of stored or derived indices. The collection of all defined indices determines the feature vector of the object set. When stored to disk, each object set has its own directory, storage policy, and a group of index files. For performance reasons, there are also cache policies per object set, and methods exist for keeping active object sets in a RAM disk.

Various services can be used by the SMS database server, such as call-outs to tghe NOLib functions (see below) or the use of extra Smalltalk processes for data analysis. The SMS server really only provides the persistency layer and cache policies for open object sets. The overall architecture is as sow\\hown in Figure 1.

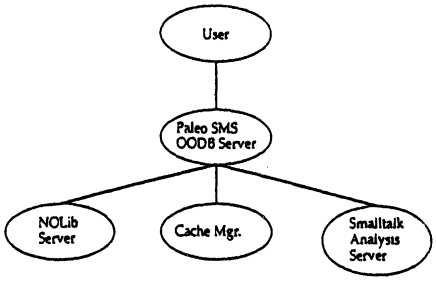

Figure 1: Paleo architecture showing the end-user API (Smoke objects) and SMS persistency server. The server uses a cache/RAM disk manager and one or more analysis back-ends

3. Paleo Media and Queries

A flexible music/sound storage facility should allow the user to manage several different kinds of data, and to interoperate with them in queries. The basic data types we use include:

- musical sound (single instrumental notes, phrases, and entire polyphonic pieces);
- non-musical sound (voice and sound effects);
- speech (single-voice utterances);
- derived or synthetic spectra and spectral families;
- musical scores (e.g., "dead-pan" or interpreted MIDI data, or notated score information); and
- expressive data (timing, amplitude, timbre) from performance ("interpretation")

This list is not meant to be exhaustive, and future additions are expected. Several query domains are implemented, such as:

- query by annotation keyword (bibliographical, analytical, text, hyper-links);
- query by sound similarity (many kinds of match);
- query by musical content or structure (phrases, harmony, formal structure);
- query by spoken phrase, language, prosody, and expression (assumes some speech recognition);
- query by instrumental performance technique (assumes capture of expressive data); and
- query by expression (assumes keyword mapping of expressive performance data).

For real-world example queries, we've worked with the following scenarios:

(Sound)

- Find an oboe note (a car crash, a reggae band).
- Find a brighter oboe note.
- Find the word "moon" whispered in Mandarin by a male voice in a dry recording.

(Spectrum)

- Find the spectrum of this note in a lower register.
- Match a given spectrum (against a set of families or templates).

(Score)

- Find Scarlatti Sonata K204.

- Find all Scarlatti sonatas in F Major.
- Find all hymns that begin with an arppeggiated chord in the tenor voice.
- Find all pieces that have plagal cadences.
- Find 7-based rhythms with the emphasis on 1, 4.

(Performance)

- Find a performance of this phrase with more rubato (faster, slower, different instruments).

4. Content, Annotation, and Indexing

The design issues related to the extent to which the index management and caching of the underlying database is visible can be reduced to a number of maxims:

- The user doesn't care what's content, what's annotation, and what's indexing.
- In theory, if we have enough annotation and indexing, we can throw the content away!
- For flexible applications, we need to be able to generate and manage annotation and indexing on-the-fly.
- For some object models, it is natural to merge content and annotation (scores), and for others, it's more natural to separate them (sound).

In building suites of analysis routines for Paleo, we have realized that there are many different groups of analyses that one might need, and that the optimal set depends on the style or genre of the (musical) data, the assumed use of the object set, and the medium of the source data. Part of the experiment is to build a broad range of object sets and applications that use them, and to see how the feature vectors evolve over time.

5. DSP Analysis with NOLib

NOLib is a suite of data analysis and feature extraction routines written by one of the authors (Orio) in the MatLab programming language. These functions can be called by analysis scripts (interpreted MatLab programs), which can themselves be started by a network-based "analysis server." We use the public-domain "octave" (Octave 1999) implementation of MatLab running on SGI (IRIX) and Apple (LinuxPPC) servers.

NOLib's main routines are introduced below. Note that MatLab functions routinely return more than one result, as implied by the syntax [a b c] = fcn(x, y, z).

Analysis

env [epos eneg]=env(x,np) — Envelope of a signal, sampled at a multiple of the period

findport [inpos endpos ltm]=findport(x,Fs) — Find the pseudo-steady portion of a signal and the logarithmic attack time

acorr ac=acorr(x) — Autocorrelation of a signal

rceps rc=rceps(x) — Real cepstrum of a signal

pitchceps [pit,mx]=pitchceps(x,Fs) — Evaluates the pitch with real cepstrum

pitchcor [pit,m]=pitchcor(x,Fs) — Evaluates the pitch with autocorrelation

peramdf [per thr]=peramdf(x,Fs,Fmin,Fmax) — Find the fundamental period of a signal using Average magnitude difference

pitamdf [pit thr y]=pitamdf(x,Fs,Fmin,Fmax) — Average magnitude difference pitch detector

pitfol pit=pitfol(x,Fs,windan,hopsize) — Basic pitch follower

pv [mod pha ifr hopsize]=pv(x,Fs,pitch,hop) — Phase vocoder

lpc_f acoef=lpc_f(x,Fs,numcoef,windan,hopsize) — Linear prediction coding

lpcdec s=lpcdec(x,Fs,numlpc,pitch) — Signal deconvolution using LPC analysis

mfcc [meanEnergy mfccMean]=mfcc(x, lenFrame, hopsize,Ff) — Mel frequency cepstral coefficients evaluation

psyan [rms cgs irr]=psyan(x,Fs,pit) — Psychoacoustical analysis of the sound, frame by frame

ansync_mfc [mfc indnote interv] = ansync_mfc (x,Fs,numcoef,windan,hopsize,pitarray,pitres) — Pitch synchronous analysis, using mel-cepstrum, of a performance

linceps_f lfc=linceps_f(x,Fs,numcoef,windan,hopsize) — Computes linear frequency cepstral coefficients of a signal (Hamming window)

pca [proj eigvec weig]=pca(coeff) — Principal component analysis

all3save [acoef lfc mfc pos] = all3save (x, Fs, numcoef, windan, hopsize, trs) — LPC + MFCC + LFCC with variable resolution

Example: Batch analysis of a set of guitar sounds

The example MatLab code below is a script that reads a directory of sound files and derives several relevant features. We used this script for 100 recorded guitar notes to build an object set for query based on performance technique

```
    # Set the sampling rate &  window dimension
Fs=48000;
windan=2048;
    # Path to the files
fpath='/home/nico/DEI/';
    # Name of the group of files plus their extension
fnames=[...list_of_file_names...];
extens='.aiff';
    # Number of files
[filnum,void]=size(fnames);
    # Number of coefficients
numcoef=20;
    # Initialization of the linceps table
lfc=zeros(20,filnum);

    # Loop for the analysis
for n=1:filnum,
        # Load the file
    filename=[fpath,deblank(fnames(n,:)),extens];
```

```
    [head x]=ldaiff(filename);

        # Find portion of file after the attack
    [inpos endpos]=findport(x,Fs);
        # Check size of "steady state"—0.1sec max
    if(endpos-inpos>Fs*0.1)
        endpos=inpos+Fs*0.1;
    endif

        # Analyze a frame with Linear Cepstrum
    tmp_lfc=linceps_f(x(inpos:endpos), Fs,
                      numcoef, windan, windan/2);
        # Compute the mean
    lfc(1:numcoef,n)=mean(tmp_lfc')';
endfor              # End of file loop

    # Principal Component Analysis
[projection eigvec weig]=pca(lfc);
```

Other NOLib scripts have been used for audio file segmentation, speaker identification, flute performance analysis, and sound effect feature extraction.

6. MIDI File Analysis with Constraints

Our purpose is to allow complex queries on various kinds of musical data, including scores, in the spirit of the Humdrum system (Huron 1994). A large amount of digitalized music is available as MIDI files, for instance on one of the MIDI archives on the Internet.

The MIDI format however, provides only low-level musical information: it is rather a performance- than an analysis-oriented representation of music. Thus, we need to analyze MIDI files in order to compute additional musical features, such as: pitch-classes (by resolving enharmonic ambiguities), voice leading, keys, and harmonies.

Preliminary Remarks

The different tasks of analysis — enharmonic, melodic, tonal, and harmonic analysis — are not independent. For instance, the enharmonic analysis depends on tonal analysis, and conversely, the computation of local keys is based on the frequency of the different pitch-classes. Therefore, we need a global strategy in which the different tasks are performed simultaneously.

In our context, we often need a partial analysis because many queries only involve a few specific elements or incomplete information. Consider the following queries: *"How many sonatas by Scarlatti end with a perfect cadenza?"* or *"Are there more minor than major chords in the preludes of Bach's WTC?"* In such cases, it is useless to perform a complete harmonic analysis of the 555 sonatas by Scarlatti, or of the 48 preludes of the WTC. This speaks for a scheme allowing partial and incomplete analysis.

What to analyze also depends on various parameters, such as the epoch, the style, and the nature (i.e. form, instrumentation) of the music considered, e.g. the anatomic limitations of human voice compared to a keyboard instrument. Our analysis strategy should be easily adaptable to various situations.

The previous remarks led us to considering an approach based on *constraint satisfaction*, instead of using specific algorithms for the different tasks on analysis (Mouton 1994). First, as a declarative paradigm, constraint satisfaction permits to build systems that can be adapted to specific situations easily. For instance, adapting the system to vocal or keyboard music analysis is just a matter of using a different set of constraints for the melodies. Besides, constraint resolution can be partial and incomplete. More precisely, the query *"How many sonatas by Scarlatti end with a perfect cadence?"* will only require the computation of elements related to the last two chords of each sonata. Finally, constraint resolution is a global process, in which the different elements are progressively computed, thus, interdependent tasks are interlaced in the resolution.

Constraint Satisfaction

A *constraint satisfaction problem* or *CSP* (Mackworth 1977) consists of a set of *variables* (each one associated with a set of possible values, its *domain*), representing the unknown values of the problem, and a set of *constraints*, expressing relationships between them. Solving a CSP consists in instantiating each variable with a value in its domain so that the constraints are satisfied.

Our approach to analyzing a MIDI file consists in the following steps. First, we quantify the MIDI file in order to get rid of slight tempo fluctuations, and we segment it into a series of positions. Then, we define a CSP, whose variables represent the different elements of analysis: notes (one for each MIDI note-event), chords (at each position), keys (at each position), and melodies, and whose constraints represent the relationships holding between them. The set of constraints depends on the style and the form of the piece. Then we solve the CSP using standard CSP resolution. We use the BackTalk (Roy 1998) constraint solver to state and solve the problem.

For instance, to find the name and octave index for a given MIDI key, we can state a CSP with 3 variables — the pitch, the octave index and the MIDI key — and a constraint stating that their values should be consistent. E.g., MIDI key 48 corresponds to C2, B#1, or Dbb2. If the local key at this position is C major, the pitch variable will be instantiated with C, and the octave index with 2 (See Figure 2).

Here are some examples of constraints:

- Two consecutive notes in a melody don't overlap
- Two consecutive notes in a melody make a perfect, minor of major interval
- Two consecutive notes in a melody are distant from less than an octave

- The range of a melody is limited (depending on the instrumentation)
- Modulation are limited to neighbor tonalities
- A note should belong to the local tonality, or at least be closed (e.g. no Cbb in C# major)
- A local key is determined by the frequency of the notes around its position

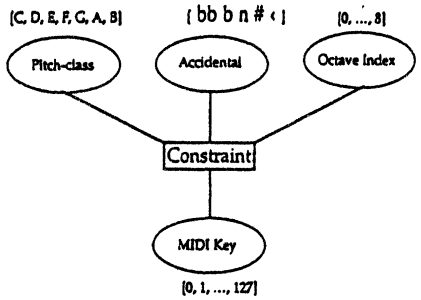

Figure 2: A CSP corresponding to the enharmonic resolution problem (the domains of the variables are in square brackets)

For each note event of the MIDI file, we define the following variables in the CSP: MIDI key, pitch, octave index, melody index, start-time, and duration. Each position corresponds to two variables: a local key, and a chord. The MIDI key, pitch, and the octave index variables are linked together as explained above. Melodic constraints link the different variables corresponding to a note (pitch, octave index, start-time and duration) as well as the melody index variables. Two melody index variables have the same value if the corresponding notes are in the same voice. Each local key variable is linked to the group of notes around its position by a constraint that computes the key according to the frequency of the different pitches. Each chord variable is linked to the note variables at its position and to the corresponding local key variable by a constraint computing the chord name and structure.

Issues

The constraint-based approach raises several issues concerning both efficiency and design. Performing the analysis as a global process can be expensive, because of its highly combinatorial nature. There are several ways of speeding up the resolution. First, some basic analysis can be performed before, and independently from, the rest. We do a pre-melodic analysis, which yields some partial melodies that are obvious in the MIDI file (e.g. in a fugue, the beginning of the first voice is easy to detect). We also perform a tonal pre-analysis, which give us information about the tonal structure of the piece, e.g. the overall tonality. This pre-analysis reduces the number of constraints and the size of the domains in the resulting CSP, which is critical for the efficiency of the resolution. The second improvement, which is standard in constraint satisfaction, is to use selection heuristics, which allows a fine control of the resolution, for instance by first focussing on easiest parts of the problem.

One difficulty is that all the aspects of analysis cannot be fairly represented as constraints. In case of ambiguities, when several solutions are valid, one of them is generally preferred to the others, because it's more likely to occur. Representing such preferences as constraints would lead to an unnecessary over-constrained problem. To address this issue, we plan to use a soft-constraints paradigm, see *semiring-based* CSP (Bistarelli 1995).

Another issue that has both design and efficiency impact is that some constraints are mere Boolean properties, which perfectly fit in the constraint satisfaction scheme, while others are more complex to handle. For instance, the last example above: *A local key is determined by the frequency of the notes around its position*, requires some substantial computations. Modern constraint satisfaction solvers permit to represent computation processes as constraints, which are efficiently handled during the resolution (Roy 1998, 1999).

7. Paleo Database Usage

To set up Paleo, we create a database given a storage directory, then create one or more object sets in it (these correspond to classes or tables), and lastly define indices for the object sets (corresponding to instance variables and accessors). One can then add object to an object set, or retrieve objects based on queries.

Create a new database of scores

The first example establishes a new database and adds an object set to it. The object we add to this set are assumed to respond to the messages *composer* and *style*. The examples that follow are in Smalltalk; comments are enclosed in double-quotes.

```
| dir db |
dir := 'Nomad:Paleo'.                "base directory"
db := SMSDB newOn: dir.              "DB object"
(db addObjectSetNamed: #Scores)
    objectsPerFile: 1;               "Add an obj-set"
    storesClass: EventList;          "Stores event lists"
                                     "Add 2 indices"
    indexOn: #composer domain: String;
    indexOn: #style domain: Symbol.

db save.                             "Save the object set"
                                     "Store objects"
db storeAll: (...collection_of_scores...)
```

Make a simple query

To make a simple database query, we re-open the database, and create a *getOne*: message with one or more *where*: clauses, e. g., to get a score by name.

```
| db |
db := MinneStoreDB openOn: 'Nomad:Paleo'.
(db getOne: #Scores)        "Create a query on name"
    where: #name eq: #ScarlattiK004;
    execute                 "Get the first response"
```

Add a new index to an existing database

To add a new index to an existing object set, we use the *indexOn:* message, giving it the name of a "getter" method (i.e., the method that answers the property of the index), or simply provide a block of Smalltalk code to execute to derive the index value. In the second part of the next example, we create an index of the pitches of the first notes in the score database using a block (the text between the square brackets) that gets the first pitches. This getter block could involve more complex code and/or calls to NOLib functions.

```
                "Add a new index with getter method"
(db objectSetNamed: #Scores)
    indexOn: #name domain: Symbol.

                "Add an index with getter block"
(db objectSetNamed: #Scores)
    indexOn: #firstPitch
    domain: SmallInteger
    getter: [ :el | el events first event
                pitch asMIDI value].
db save.
```

Make a more sophisticated query

To retrieve objects from the database, we use *getOne:* or *getAll:* as above, and can, for example, ask for a range or the derived first-pitch feature.

```
(db getAll: #Scores)
    where: #firstPitch between: 62 and: 65;
    execute
```

8. Compact and Efficient Storage Formats

Paleo supports compact and efficient data I/O in the form of methods that work with the Squeak Smalltalk *ReferenceStream* framework, a customizable binary object streaming format. The trade-offs in the design of object storage formats are between size, complexity, and flexibility (pick any two). In Paleo, we opted for a system that is compact but also supports the full flexibility of the Smoke music representation, including abstract models for pitch, time, and dynamics, multiple levels of properties and annotation, the attachment of functions of time to events, and hyper-links between events or event lists.

Below is a hexadecimal dump of a simple event list (score) that demonstrates mixed-type properties, i.e., the pitches are mixed among MIDI key numbers, note names, and Hertz values. The format shown here is the "debugging format" and has some extra key bytes and explicit class names rather than more compact symbol table keys.

```
Hex data bytes      Description
09 00 00 00 06      Object header
                    key 09 = normal object
                    32-bit inst size 6 = 4 instVars
                    + 1 property + 1 offset
06 09 45 76 65 6E 74 4C 69 73 74
                    Object type: class name,
                    key 06 = String, size = 09,
                    value = 'EventList'
80 58 04 00 00 08 6E

                    Event list header, properties
                    (dur) key 80 = music magnitude,
                    key 58 = MSecondDuration,
                    key 04 = smallInt,
                    32-bit int value = 08 6E msec
01 01 01                nil pitch, ampl, voice
06 04 6E 61 6D 65       property key = 'name'
06 03 74 65 36          property value = 'te6'
00 00 00 0D             # of events (13)
                    End of header, note data follows

C0 80 58 04 00 00 00 00
                    Note event header, C0 = event key,
                    start time, 80 = music mag. key
                    58 = msec dur key, 04 = smallInt,
                    32-bit val = 0
00 00 00 05         obj size = 4 instVars + 0 props + 1
06 0A 4D 75 73 69 63 45 76 65 6E 74
                    class name, key 06 = String,
                    size = 0A, = 'MusicEvent'
80 58 04 00 00 00 A6
                    duration, key 58 = MSec dur,
                    key 04 = smallInt,
                    32-bit int value A6 = 166 msec
80 5C 11 02 63 32
                    pitch, key 5c = SymbolicPitch,
                    key 11 = string,
                    length = 2, string = 'c2'
80 47 04 00 00 00 48
                    loudness, key 47 = MIDIVelocity
01                  voice = nil, End of event

C0 80 58 04 00 00 00 A6
                    Event header, (start time) music
                    magnitude, msec dur
00 00 00 05         obj size = 4 iVars + 0 props + 1
0A 00 00 00 35      reference to class name
80 48 0E 3F C5 3F 7C
                    duration, key 48 = SecondDuration,
                    key 0E = float
80 5C 11 03 63 23 32
                    pitch, key 5c = SymbolicPitch,
                    key 11 = string...
80 54 0E C0 13 B7 BB
                    loudness, key 47 = DBLoudness,
                key 0E = float
01              voice = nil, End of event

                —more events follow
```

Data files in this format is on the order of 10-40 times larger than the "corresponding" MIDI files, but because this notation supports the full Smoke annotation, we can store much richer data. Paleo extensions include simple derived properties such as symbolic pitch (with enharmonic disambiguation) and time (with tempo and meter derivation, rest insertion, and metrical grouping), and higher-level properties such as harmonic analysis, performance expression, and others.

9. Issues and Questions

The sections above have brought to light several design dimensions in Paleo; these are each the subjects of on-going evaluation and redesign.

There are unsolved problems in the basic object modelling area, such as how to model and manipulate derived performance expression—as tempo and dynamic maps; as the weights of rule sets; as constraint filters, etc.

As mentioned above, we are experimenting with the association of analysis suites with families of object sets so that we can classify new data according to its style and usage to determine what first-round analysis to undertake.

Other questions relate to the incorporation of new analysis methods (e.g., based on the mapping of performance rules onto real performances) and the use of these in queries.

10. Conclusion

While we do have many interesting intermediate results, Paleo is still very much a work in progress.

The planned extensions will make the system even more unique and powerful for a wide range of sound and music applications.

Squeak, Siren, SMS, NOLib, and Paleo are all available from the CREATE web/ftp site. See http://www.create.ucsb.edu/Paleo for details. Most of the software runs on Macintosh, MSWindows, and UNIX platforms and is available with source code in C, MatLab, or Smalltalk.

References

Bistarelli, S., U. Montanari, and F. Rossi. 1995 "Constraint Solving over Semirings," in Proceedings of IJCAI'95

Carlson, J. 1998. MinneStore OO Database Documentation. See http://www.objectcomposition.com.

Huron, D. 1994. "The Humdrum Toolkit Reference Manual," Center for Computer Assisted Research in the Humanities, Menlo Park, California.

Mackworth, A. 1977. "Consistency in Networks of Relations," Artificial intelligence, 8 (1), pp. 99–118.

Mouton, R. 1994. "Outils intelligents pour les musicologues," Ph.D. Thesis, Université du Maine, Le Mans, France.

Octave Programming Language. See http://www.che.wisc.edu/octave/

Pope, S. T. 1997a. "Musical Object Representation." in Roads, C, S. Pope, A. Piccialli, and G. De Poli, eds. *Musical Signal Processing*. Lisse, the Netherlands: Swets & Zeitlinger, 1997. pp. 317-348.

Pope, S. T. 1997b. "Siren: Software for Music composition and Performance in Squeak." *Proceedings of the 1997 International Computer Music Conference*. San Francisco, International Computer Music Association. pp. 208-210.

Pope, S. T. 1998. "The Siren Music/Sound Platform." *Proceedings of the 1998 ACM Conference on Object Oriented Programming Systems, Languages, and Applications* (OOPSLA).

Roy, P., A. Liret, and F. Pachet. 1998. "The Framework Approach for Constraint Satisfaction," ACM Computing Survey Symposium, 1998

Roy, P., A. Liret, and F. Pachet. 1999. "Constraint Satisfaction Frameworks," in "Object-Oriented Application Frameworks," Wiley Ed., chapter 17, to appear

Squeak, 1999. Squeak Smalltalk documentation. See http://www.squeak.org.